Atomic Active Messages: A Hardware-supported Mechanism for Accelerating Graph Analytics

Maciej Besta
maciej.besta@inf.ethz.ch
ETH Zurich

Torsten Hoefler
htor@inf.ethz.ch
ETH Zurich

Abstract

We propose Atomic Active Messages (AAM), a mechanism that accelerates graph computations on both shared- and distributed-memory machines. The key idea behind AAM is that hardware transactional memory (HTM) can be used for simple and high performance processing of irregular structures in highly parallel environments. We illustrate techniques such as coarsening and coalescing that enable hardware transactions to achieve considerable speedups in graph processing. We conduct a detailed performance analysis of AAM on Intel Haswell and IBM Blue Gene/Q and we illustrate various performance tradeoffs between different HTM parameters that impact the efficiency of graph processing. AAM can be used to implement abstractions offered by existing programming models and to improve the performance of graph analytics codes such as Graph500 or Galois.

1 Introduction

Big graphs stand behind many computational problems in social network analysis, machine-learning, computational science, and others [23]. Yet, designing efficient parallel graph algorithms is challenging due to intricate properties of graph computations. First, they are often *data-driven* and *unstructured*, making parallelism based on partitioning of data difficult to express. Second, they are usually *fine-grained* and have *poor locality*. Finally, implementing synchronization based on locks or atomics is tedious, error prone, and typically requires concurrency specialists [23].

Recent implementations of hardware transactional memory (HTM) [12] promise a faster and simpler programming for parallel algorithms. The key functionality is that complex instructions or instruction sequences execute in *isolation* and become visible to other threads *atomically*. Available HTM implementations show promising performance in scientific codes and industrial benchmarks [37, 33]. In this work, we show that the ease of programming and performance benefits are even more promising for fine-grained, irregular, and data-driven graph computations.

Another challenge of graph analytics is the size of the input that often requires distributed memory machines [24]. Such machines generally contain manycore compute nodes that may support HTM (cf. IBM Blue Gene/Q [33]). Still, it is unclear how to handle transactions accessing vertices on both local and remote nodes.

In this paper we propose a mechanism called *Atomic Active Messages* (AAM) that accelerates graph analytics by combining the active messaging (AM) model [32] with HTM. In AAM, fine units of graph computation (e.g., marking a vertex in BFS) are *coarsened* and executed as hardware transactions. While software-based coarsening was proposed in the past [16], in this paper we focus on developing high performance hardware-supported techniques to implement this mechanism on both shared- and distributed-memory machines, on establishing principles and practice of the use of HTM for the processing of graphs, and on illustrating various performance tradeoffs between different HTM parameters in the context of graph analytics. Figure 1 motivates AAM by showing the time to perform each phase in a synchronized BFS traversal using traditional fine-grained atomics and AAM based on coarser hardware transactions.

Another key insight of our work is that AAM constitutes a hierarchy of atomic actives messages that can be used to accelerate graph computations on both shared- and distributed-memory machines. We analyze this hierarchy in detail and conclude that AAM can be used to improve the performance of generic graph analytics tools such as Galois or Graph500. The key contributions of our work are:

- We design the generic AAM mechanism that uses state-of-the-art HTM implementations to accelerate both shared- and distributed-memory graph computations.

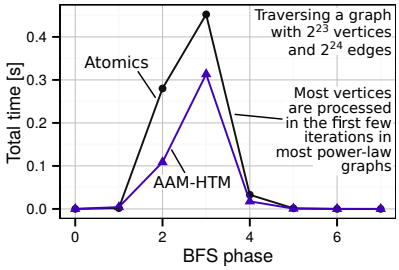


Figure 1: Comparison of the duration of an intra-node BFS traversal implemented with Blue Gene/Q fine-grained atomics and coarse hardware transactions (AAM-HTM). One transaction modifies 2^7 vertices. We use 64 threads and a Kronecker graph [21] with a power-law vertex degree distribution.

- We establish the principles and practice of the use of HTM for graph computations. Specifically, we develop protocols for spawning remote/distributed hardware transactions.
- We introduce a performance model and we conduct a detailed performance analysis of AAM based on Intel Haswell HTM [37] and IBM Blue Gene/Q HTM [33] to illustrate various performance tradeoffs between different HTM parameters in the context of graph analytics. Specifically, we find optimum transaction sizes for x86 and PowerPC machines that accelerate Graph500 [27] BFS code by >100%.
- We show that AAM can be seamlessly used to accelerate state-of-the-art graph analytics engines for the processing of various synthetic and real-world graphs.

2 Background

We now describe active messages, atomics, transactional memory, and how they are used in graph computations.

2.1 Active Messages

In the active messaging (AM) model [32] processes exchange messages that carry: the address of a user-level handler function, handler parameters, and optional payload. When a message arrives, the parameters and the payload are extracted from the network and the related handler runs at the receiver [35]. Thus, AMs are conceptually similar to lightweight Remote Procedure Calls (RPCs).

Active messages are often used to implement low-level performance-centric libraries that serve as a basis for developing higher-level libraries and runtime systems. Example libraries are Myrinet Express (MX), IBM's Deep Computing Messaging Framework (DCMF) for BlueGene/P, IBM's Parallel Active Message Interface (PAMI) for BlueGene/Q, GASNet [2], and AM++ [35].

2.2 Active Messages in Graph Computations

A challenging part of designing a distributed graph algorithm is managing its data flow. One way is to use a distributed data structure (e.g., a distributed queue) that spans all of its intra-node instances. Such structures are often hard to construct and debug [8]. A BFS algorithm that uses a distributed queue is presented in Listing 1.

```
if (source is local) Q.push(source);
while (!Q.empty()) {
  for (Vertex v : Q)
    if (v.visited == false) {
      v.visited = true;
      for (Vertex w : v.neighbors()) {Q.add(w); } } }
```

Listing 1: Distributed BFS using a distributed queue [10] (§ 2.2)

Another approach uses active messages to express the data flow of the program dynamically. When a process schedules computation for a vertex, it first checks whether it is the *owner* of this vertex. If yes, it performs the computation. Otherwise, the computation is sent in an active message to a different node for processing in a remote handler [36]. Thus, no distributed data structures have to be used. We illustrate BFS using this approach in Listing 2.

```
struct bfs_AM_handler {
  bool operator()(const pair<Vertex, int>& x) {
    if (x.second < x.first.distance) {
      x.first.distance = x.second;
      send_active_message(x.first, x.second + 1); } }
          };
```

Listing 2: Distributed BFS using active messages [36] (§ 2.2)

2.3 Atomic Operations

Atomic operations appear to the rest of the system as if they occur instantaneously. Atomics are used in lock-free graph computations to perform fine-grained updates [10, 27]. Yet, they are limited to a single word and thus require complex protocols for protecting operations involving multiple words. We now present relevant atomics:

Accumulate(*target, arg, op) (ACC): it applies an operation op (e.g., sum) to *target using an argument arg.

Fetch-and-Op(*target, arg, op) (FAO): similar to `Accumulate` but it also returns the previous value of `*target`.

Compare-and-Swap(*target, compare, value, *result) (CAS): if `*target == compare` then `value` is written into `*target` and the function sets `*result` to `true`, otherwise it does not change `*target` and sets `*result` to `false`.

2.4 Transactional Memory

Transactional Memory (TM) [12] is a technique in which portions of code (*transactions*) are executed in isolation and their memory effects become visible atomically. Thus, such code portions are linearizable and easy to reason about. The underlying TM mechanism detects dependencies between transactions accessing the same memory locations and solves any potential *conflicts* between such accesses. It must also record all modifications to specific memory locations and then commit them atomically. TM can be based on software emulation [31] (software transactional memory; STM) or native hardware support [12] (HTM).

Several vendors introduced HTM implementations: IBM, Sun, and Azul added HTM to Blue Gene/Q (BG/Q) machines [33], the Rock processor [6], and the Vega CPUs [5], respectively. Intel implemented two HTM instruction sets in the Haswell processor: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) that together constitute Transactional Synchronization Extensions (TSX) [37]. HLE allows for fast and simple porting of legacy lock-based code into code that uses TM. RTM enables programmers to define transactional regions in a more flexible manner than that possible with HLE [37].

There are few existing studies on STM in graph computations [14]. Using HTM in graph processing has been largely unaddressed and only a few initial works exist [7, 34].

3 Atomic Active Messages

Atomic Active Messages (AAM) is a mechanism motivated by recent advances in deploying transactional memory in hardware. An atomic active message is a message that, upon its arrival, executes a user-specified handler called an *operator*. A *spawner* is a process (or a thread within this process, depending on the context) that issues atomic active messages. An *activity* is the computation that takes place as a result of executing an operator. Activities run speculatively, isolated from one another, and they either *commit* atomically or do not commit at all. We distinguish between operators and the activities to keep our discussion generic.

To use AAM, the developer specifies the operator code that modifies elements (vertices or edges) of the graph. We use single-element operators for easy and intuitive programming of graph algorithms. Still, multiple-element *coarse* operators can be specified by experienced users. The developer also determines the structure of a vertex or an edge and defines the *failure handler*, an additional piece of code executed in certain types of algorithms (explained in § 3.2).

Our runtime system executes algorithms by exchanging messages, spawning activities to run the operator code, running failure handlers, and optimizing the execution. An activity can be *coarse*: it may execute *several* operators atomically. Note that operators are (optionally) coarsened by the developer while activities are coarsened by the runtime.

The implementation determines how activities are isolated from one another. An activity can execute as a critical section guarded by locks, or (if it modifies one element) as an atomic operation (e.g., CAS in BFS). However, we argue that in many cases running activities as hardware transactions provides the highest speedup; we support this claim with a detailed performance study in Sections 5 and 6.

3.1 Definitions and Notation

Assume there are N processes $p_1, ..., p_N$ in the system. A process p_i runs on a compute node $n_i, 1 \geq i \geq N$ and it may contain up to T threads. Then, we model the analyzed graph G as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges between vertices. Without loss of generality we assume that G is partitioned and distributed using a one-dimensional scheme [4]: V is divided into N subsets V_i and every $V_i \subseteq V$ is stored on node n_i. We call process p_i the *owner* of every vertex $v \in V_i$ and every edge (v, w) such that $v \in V_i, w \in V$. We denote the average degree in G as $\bar{d}$.

3.2 Types of Atomic Active Messages

AAM accelerates graph computations that run on a single ($N = 1$) or multiple ($N > 1$) nodes. If $N = 1$ then messages only spawn intra-node activities. If $N > 1$ then a message may also be sent over the network to execute a remote activity. Now, we identify two further key criteria of categorizing messages: *direction of data flow* and *activity commits*. They enable four types of messages; each type improves the performance of different graph algorithms.

3.2.1 Direction of Data Flow

This criteria determines if an activity has to communicate some data back to its spawner. In some graph algorithms the data flow is *unidirectional* and messages are *Fire-and-Forget* (FF): they start activities that do not return any data. Other algorithms require the activity to return some data to the spawner to run a failure handler. We name a message that executes such an activity a *Fire-and-Return* (FR) message (the flow of data is *bidirectional*).

3.2.2 Activity Commits

In some graph algorithms messages belong to the type *Always-Succeed* (AS): they spawn activities that have to successfully commit, even if it requires multiple rollbacks or serialized execution. An example such algorithm is PageRank [3] where each vertex v has a parameter *rank* that is augmented with the normalized ranks of v's neighbors. Now, if we implement activities with transactions, then such transactions may conflict while concurrently updating the rank of the same vertex v, but *finally each of them has to succeed* to add its normalized rank. The other type are *May-Fail* (MF) messages that spawn activities that may also fail ultimately and not reexecute after a rollback. An example is BFS in which two activities, which concurrently change the distance of the same vertex, conflict and only one of them succeeds. Note that we distinguish between rollbacks of activities at the algorithm level, and aborts of transactions due to cache eviction, context switches, and other reasons caused by hardware/OS. In the latter case the transaction is reexecuted by the runtime to ensure correctness.

Our criteria entail four message types: FF&AS, FF&MF, FR&AS, FR&MF. We now show examples on how each of these types can be used to program graph algorithms.

3.3 Example Case Studies

In AAM, a single graph algorithm uses only one type of atomic active messages. This type determines the form of the related operator and the existence of the failure handler. Here, we focus on the operator as the most complex part of graph algorithms. We show C-like code to implement the operator in isolation. Our implementation utilizes system annotations to mark atomic regions in C. We present the code of five single-element operators. When necessary, we discuss the failure handlers. We describe multiple-element operators at the end of this section.

3.3.1 PageRank (FF & AS)

PageRank (PR) [3] is an iterative algorithm that calculates the *rank* of each vertex $v \in V$: $rank(v) = \frac{1-d}{|V|} + \sum_{w \in n(v)}(d \cdot \frac{rank(w)}{out\_deg(w)})$. $n(v)$ is the set of v's neighbors, d is the *dump factor* [3] and $out\_deg(w)$ is the number of links leaving w. Depending on the operator design, PR may be either *vertex-centric* and *edge-centric*.

The pseudocode of the vertex-centric variant is presented in Listing 3. The operator increases the ranks of v's neighbors with a factor $d \cdot \frac{rank(v)}{out\_deg(v)}$. It also adds $\frac{1-d}{|V|}$ to $rank(v)$. The copies of stale ranks from a previous iteration are kept and used for calculating new ranks. Assuming that each vertex v is processed by one activity, this PR variant uses AS messages: each activity has to successfully add the factors to the ranks of respective vertices (which may require serialization). Data flow is unidirectional (messages are FF) because activities do not have to communicate any results back to their spawners. Thus, the operator returns `void`.

```
void Operator(Vertex v) {
  v.rank += (1 - d) / vertices_nr;
  for(int i = 0; i < v.neighbors.length; i++) {
    v.neighbors[i].rank += d * v.old_rank/v.out_deg;
        } }
```

Listing 3: The operator in the vertex-centric PageRank variant (§ 3.3.1)

There exist other PR variants. Specifically, one can analyze incoming edges to dispose of conflicts. We will later (Section 6) show that a careful AAM design outperforms such approaches used in various codes such as PBGL.

3.3.2 Breadth First Search (FF & MF)

Breadth First Search (BFS) uses FF & MF messages. Spawners do not have to wait for any results, but some activities may fail when concurrently updating vertices using different distance values. Such a conflict is solved at the node owning the vertex and no information has to be sent back to any of the spawners, thus the operator returns `void`. We present the operator pseudocode in Listing 4.

```
void Operator(Vertex v, int new_distance) {
  if(v.distance > new_dist) {v.distance = new_dist;}
      }
```

Listing 4: BFS operator (§ 3.3.2)

3.3.3 Boruvka Minimum Spanning Tree (FF & MF)

Boruvka is an algorithm for finding the minimum spanning tree of a graph with weighted edges. First, a *supervertex* is created out of each vertex. The activity finds the smallest-weight edge incident to each accessed supervertex, merges two supervertices connected with this edge, and appropriately modifies the remaining incident edges. Boruvka uses May-Fail messages: if two concurrent activities conflict, then one of them will fail. Messages are also Fire-and-Return as the activity has to communicate back whether it succeeded or failed, so that the spawner may choose to either retry or, e.g., backoff for some time. Listing 5 illustrates the operator pseudocode.

```
void Operator(Supervertex v) {
        Edge edge = get_smallest_weight_edge(v);

        forall(Edge e: get_incident_edges(edge.dest))
              {
                if(e.dest == v) continue;
                //set 'v' as a new destination for e
                e.change_edge_dest(v);
        }
        edge.dest.delete();
}
```

Listing 5: Boruvka operator (§ 3.3.3)

3.3.4 ST Connectivity (FR & AS)

ST connectivity [28] determines if two given vertices (s and t) are connected. First, the algorithm marks each vertex as "white". Then, it starts two concurrent BFS traversals from s and t. Both traversals use different colors ("grey" and "green") to mark vertices as visited. Each activity returns the information on the colors of visited vertices. In case of "white" no action is taken and the operator returns `false`. If the found color is used by the other BFS, then s and t are connected, the operator returns `true`, and the runtime executes a failure handler at the spawner that terminates the algorithm. The operator is presented in Listing 6.

```
bool Operator(Vertex v, Color new_col) {
  if(v.color != WHITE && v.color != new_col) return
       true;
  v.color = new_col; return false; }
```

Listing 6: ST Connectivity operator (§ 3.3.4)

3.3.5 Boman Graph Coloring (FR & MF)

Graph coloring proposed by Boman et al. [1] is a heuristic algorithm that minimizes the number of colors assigned to graph vertices. In this algorithm as expressed using AAM (see Listing 7), an activity changes the color or vertex v to X. Then, if any of v's neighbors has color X, either v or the neighbor has to change its color; the choice is random. Activities are spawned by MF & FR messages because multiple processes trying to update v's color may conflict and the spawners have to be notified if they need to assign new colors to v's neighbors in failure handlers.

```
int Operator(Vertex v, Color X) {
  v.Color = X;
  if(v.hasNeighborWithColor(X)) {
    //return the ID of a vertex to be recolored
    if(rand([0;1]) < 0.5) return v.neighborWithCol(X)
        .ID;
    else return v.ID;
  } else {  //NO_VERTEX_ID means no vertex is
      recolored
    return NO_VERTEX_ID; }
```

Listing 7: Boman graph coloring operator (§ 3.3.5)

3.4 Discussion

The introduced AAM operators modify single vertices. Thus, they enable intuitive developing and reasoning about graph computations that are also fine-grained by nature. Still, some users may want to specify coarser operators to use additional knowledge that they have about the graph structure for higher performance. Here, the user determines the number of elements to be modified in the operator and the policy of their selection (e.g., the operator may choose each vertex randomly, or try to modify elements stored in a contiguous block of memory to avoid HTM aborts).

Manual coarsening of operators may be challenging. Our runtime system automatically coarsens activities for easier AAM programming. We now discuss the implementation details of coarsening and other optimizations. While single-element operators can be implemented with atomics or fine-grained locks, we argue that a more performant approach is based on coarse transactions.

4 Implementing Activities

We now discuss the details of implementing activities; we skip most of the issues related to the runtime as they were properly addressed in other studies [35, 8, 36].

4.1 Implementing Activities with HTM

In this paper we advocate for using HTM to implement activities. However, locks and atomics would also match the activity semantics (atomics can implement fine activities that modify single words). We thus compared the performance of all the three mechanisms to illustrate HTM's advantages.

Transactions can implement an activity of any size. We use Intel Haswell HLE and RTM ISAs[1] and IBM BG/Q HTM. RTM provides two key functions: `XBEGIN` that starts a transaction and `XEND` that performs a commit. Yet, it does not guarantee progress. Thus, we repeat aborted transactions and we use exponential backoff to avoid livelock. The HTM in BG/Q automatically retries aborted transactions and it serializes the execution when the number of retries is equal to a certain value; we use the default value (10). HLE performs serialization after the first abort. We illustrate the utilized compiler HTM intrinsics for implementing activities in Listing 8.

```
/**************** IBM BG/Q HTM ********************/
#pragma tm_atomic {
  Activity(vertices, new_distance); //run
       speculatively }

/**************** Intel RTM HTM *******************/
bool committed = false;
while(!committed) {
  /* We start an RTM transaction with _xbegin().
  XBEGIN_STARTED indicates that it began successfully
       */
  if((status = _xbegin ()) == _XBEGIN_STARTED) {
    Activity(vertices, new_distance); //run
         speculatively
    _xend (); //commit the transaction
  } else { //an abort occured
    exponential_backoff(); continue;  //rollback
  } committed = true; }

/**************** Intel HLE HTM *******************/
// Acquire lock with lock elision
while (__atomic_exchange_n(&lockvar, 1,
     __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE)) {
  _mm_pause(); /* fallback path */ }
  Activity(vertices, new_distance); //run
       speculatively
__atomic_store_n(&lockvar, 0, __ATOMIC_RELEASE|
     __ATOMIC_HLE_RELEASE);
```

Listing 8: Implementing BFS activities with HTM (§ 4.1).

4.2 Optimizing the Execution of Activities

Two most significant optimizations applied by the runtime are *coarsening* and *coalescing* of activities. First, in the intra-node computations, the runtime coarsens activities by atomically executing more than one operator; an example is presented in Listing 9. We denote activities that are not coarse as *fine*. Coarsening amortizes the overhead of starting and committing an activity; it also reduces the amount of fine-grained synchronization. Second, activities targeted at the same remote node are sent in a single message, i.e., coalesced. This reduces the overhead of sending and receiving an atomic active message and saves bandwidth. Finally, we also use various optimizations that attempt to reduce the amount of synchronization even further. For example, the runtime avoids executing the BFS operator for each vertex by verifying if the vertex has already been visited.

[1]We verify the correctness of all the results to ensure that the limitations of TSX [13] do not affect our evaluation and the conclusions drawn.

```
void Activity(Vertex vertices[], int new_distance) {
  forall(Vertex v: vertices) {
    //call the BFS operator from Listing 5
    Operator(v, new_distance); } }
```

Listing 9: A BFS coarse activity (§ 4.2)

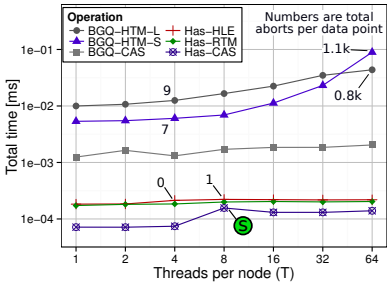
4.3 A Protocol for Distributed Activities

The *ownership protocol* enables activities implemented as hardware transactions that access or modify data from remote nodes. The basic idea behind the protocol is that a handler running such an activity has to first physically relocate all required vertices/edges to the memory of the node where the activity executes. This approach is dictated by the fact that a hardware transaction cannot simply send a message because it would not be able to rollback remote changes that this message caused. In addition, most HTM implementations prevent many types of operations (e.g., system calls) from being executed inside a transaction [33].

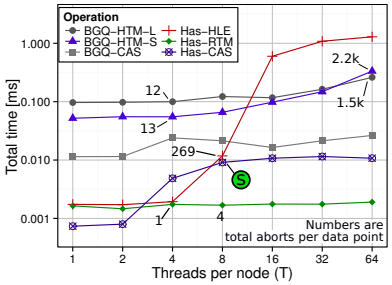
Our protocol assumes that each graph element has an *ownership marker* that can be modified atomically by any process. Each marker is initially set to a value $\perp$ different from any process id. When a transaction from a node n_i accesses a remote graph element, it aborts and the runtime uses CAS or a different mechanism (e.g., an active message) to set the marker of this element to the id of process p_i. If the CAS succeeds, the marked element is transferred to node n_i and the transaction restarts. If the CAS fails, the handler sets all previously marked elements to $\perp$ and backs off for a random amount of time. If a local transaction attempts to access a marked element, it aborts. This mechanism is repeated until all remote elements are cached locally. Finally, after the transaction succeeds, the elements are sent back to their original nodes and their markers are set to $\perp$.

5 Performance Model & Analysis

We now introduce a simple performance model that shows the tradeoffs between atomics and HTM. Then, we analyze the performance of AAM and answer the following research questions: (1) what are HTM's advantages over atomics for implementing AAM activities, (2) what are performance tradeoffs related to various HTM parameters, and (3) what are the optimum transaction sizes for analyzed architectures that enable highest speedups in selected graph algorithms.



(a) Marking a vertex as visited 10 times (§ 5.4.1).

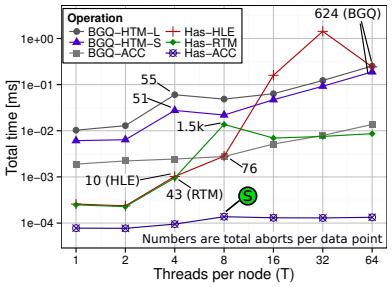


(b) Marking a vertex as visited 100 times (§ 5.4.1).

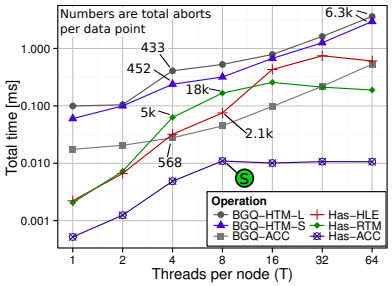
Aborts due to:

		Memory conflicts	Buffer overflows	Other reasons
10 ops	Has-RTM	2	2	0
	BGQ-HTM-L	802	3	1
	BGQ-HTM-S	1,118	46	180
100 ops	Has-RTM	2	2	0
	BGQ-HTM-L	1,539	5	1
	BGQ-HTM-S	2,242	13	2

(c) Marking a vertex as visited: details of the aborts in the HTM implementations (§ 5.4.1).



(d) Incrementing a vertex' rank 10 times (§ 5.4.2).



(e) Incrementing a vertex' rank 100 times (§ 5.4.2).

Aborts due to:

		Memory conflicts	Buffer overflows	Other reasons
10 ops	Has-RTM	1,520	1	0
	BGQ-HTM-L	624	62	614
	BGQ-HTM-S	623	62	613
100 ops	Has-RTM	18,952	33	0
	BGQ-HTM-L	6,374	637	6,360
	BGQ-HTM-S	6,392	639	6,380

(f) Incrementing a vertex' rank: details of the aborts in the HTM implementations (§ 5.4.2).

Figure 2: The analysis of the performance of intra-node activities implemented with atomics and HTMs (§ 5.4). Figures 2a and 2b illustrate the time it takes to mark a vertex as visited. Numbers in figures are sums of HTM aborts for a given datapoint (we report values for $T = 4$ for Has/BGQ; we also plot numbers for $T = 8$ (Has) and $T = 64$ (BGQ) to illustrate the numbers of aborts generated by all the supported hardware threads). Ⓢ indicates the point where the latency of Haswell atomics stops to grow. Table 2c shows the distribution of the reasons of aborts for $T = 64$ (BGQ) and $T = 8$ (Haswell). We skip Has-HLE as it does not provide functions to gather such statistics. A similar performance analysis for incrementing the rank of a vertex is presented in Figures 2d-2e and Table 2f.

5.1 Experimental Setup

We compile the code with gcc-4.8 (on Haswell) and with IBM XLC v12.1 (on BG/Q). We use the following machines:

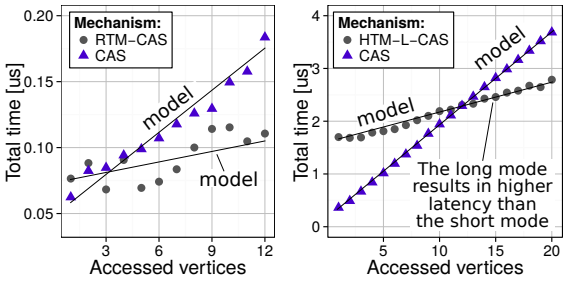
ALCF BG/Q Vesta (BGQ) is a supercomputing machine where each compute node contains 16 1.6 GHz PowerPC A2 4-way multi-threaded cores, giving the total of 64 hardware threads per node. Each core has 16 kB of L1 cache. Every node has 32 MB of shared L2 cache and 16 GB of RAM. Nodes are connected with a 5D proprietary torus network. This machine represents massively parallel supercomputers with HTM implemented in the shared last-level cache.

Trivium V70.05 (Has-C) is a commodity off-the-shelf server where the processor (Intel Core i7-4770) contains 4 3.4 GHz Haswell 2-way multi-threaded cores, giving the total of 8 hardware threads. Each core has 32 KB of L1 and 256 KB of L2 cache. The CPU has 8 MB of shared L3 cache and 8 GB of RAM. This option speaks for commodity computers with HTM operating in private caches.

Greina (Has-P) is a high-performance cluster that contains two nodes connected with InfiniBand FDR fabric. Each node hosts an Intel Xeon CPU E5-2680 CPU with 12 2-way 2.50GHz multi-threaded cores; the total of 24 hardware threads. Each core contains 64 KB of L1 and 256 KB of L2 cache. The CPU has 30 MB of shared L3 cache and 66 GB of RAM. This machine represents high-performance clusters deploying HTM in private caches.

5.2 Considered Hardware Mechanisms

For Haswell we compare the following mechanisms: RTM (`Has-RTM`), HLE (`Has-HLE`), GCC `__sync_bool_compare_and_swap` (`Has-CAS`), and GCC `__sync_add_and_fetch` (`Has-ACC`). We select CAS and ACC because they can be used in miscellaneous graph codes such as BFS (a FF&MF algorithm), PR (a FF&AS algorithm), and ST Connectivity (a FR&AS algorithm) [27]. For BG/Q we analyze: IBM XLC `__compare_and_swap` (`BGQ-CAS`) and GCC `__sync_add_and_fetch` (`BGQ-ACC`). We compare two modes of HTM in BG/Q: the *short running mode* [33] (`BGQ-HTM-S`) that bypasses L1 cache and performs better for shorter transactions, and the *long running mode* [33] (`BGQ-HTM-L`) that keeps speculative states in L1 and is better suited for longer transactions [33].



(a) Has-C, RTM. (b) BGQ, the long mode.

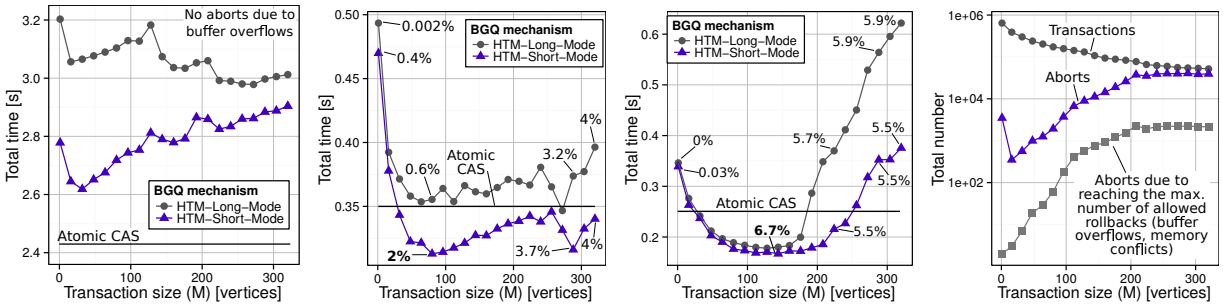
Figure 4: (§ 5.3) The validation of the performance model.

5.3 Performance Model

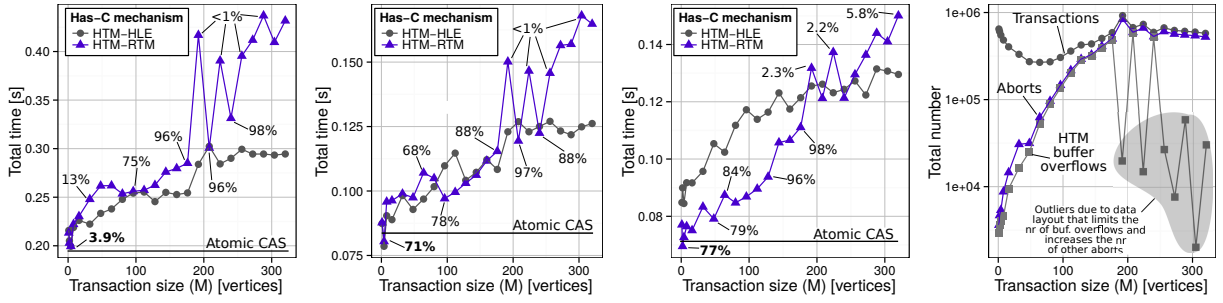
Our performance model targets graph processing and we argue in terms of activities and accessed vertices. We predict that an activity implemented as a transaction that modifies one vertex is more computationally expensive than an equivalent single atomic. Yet, the transactional overheads (starting and committing) may be amortized with coarser transactions and respective activities would outperform a series of atomics for a certain number of accessed vertices.

We now model the performance to determine the existence of crossing points; out model includes both the execution of the operations and fetching the operands from the memory. The total time to execute an activity that modifies N vertices (using either atomics or HTM) can be modeled with a simple linear function with N as the argument. We denote the slope and the intercept parameters of a function that targets atomics as $\mathscr{A}_{AT}$ and $\mathscr{B}_{AT}$; the respective parameters for HTM are $\mathscr{A}_{HTM}$ and $\mathscr{B}_{HTM}$. We predict that $\mathscr{B}_{HTM} > \mathscr{B}_{AT}$ due to high transactional overheads. On the contrary, we conjecture that $\mathscr{A}_{HTM} < \mathscr{A}_{AT}$ because HTM overheads will grow at a significantly lower rate (determined by accesses to the memory subsystem) than that of atomics.

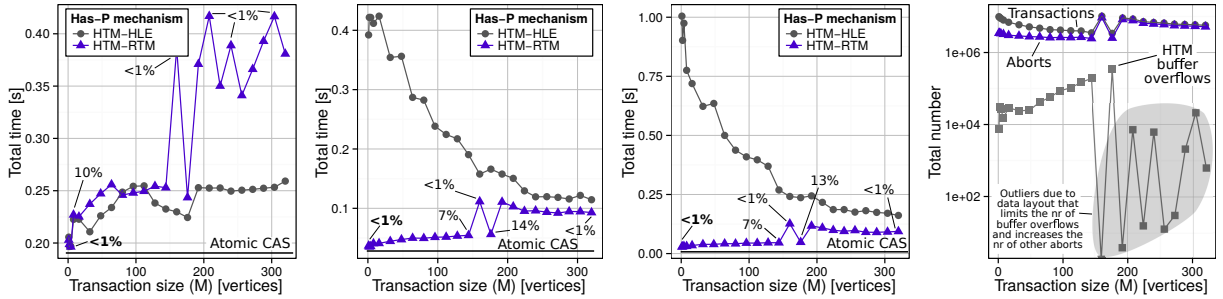
We illustrate the model validation for `CAS` in Figure 4; we plot only the results for RTM on `Has-C` and the long mode HTM on `BGQ` because all the other results differ marginally and follow similar performance patterns. We use linear regression to calculate $\mathscr{A}_{AT}$, $\mathscr{B}_{AT}$, $\mathscr{A}_{HTM}$, and $\mathscr{B}_{HTM}$. The analysis indicates that the model matches the data. While a more extended model is beyond the scope of this paper, our analysis illustrates that it is possible to amortize the transactional overhead with coarser activities. We now proceed to a performance analysis that illustrates various tradeoffs between respective HTM parameters.



(a) BFS runtime ($T = 1$) (§ 5.5.1). (b) BFS runtime ($T = 16$) (§ 5.5.1). (c) BFS runtime ($T = 64$) (§ 5.5.1). (d) BG/Q events ($T = 64$) (§ 5.5.1).



(e) BFS runtime ($T = 1$) (§ 5.5.2). (f) BFS runtime ($T = 4$) (§ 5.5.2). (g) BFS runtime ($T = 8$) (§ 5.5.2). (h) Has-C events ($T = 8$) (§ 5.5.2).



(i) BFS runtime ($T = 1$) (§ 5.5.3). (j) BFS runtime ($T = 12$) (§ 5.5.3). (k) BFS runtime ($T = 24$) (§ 5.5.3). (l) Has-P events ($T = 24$) (§ 5.5.3).

Figure 3: (§ 5.5) The analysis of the performance of Graph500 OpenMP BFS implemented with hardware transactions on BGQ (Figures 3a-3d), Has-C (Figures 3e-3h), and Has-P (Figures 3i-3l). In each figure we vary the size of the transactions `M` (i.e., the number of vertices visited). We also present the results for BFS implemented with atomics (horizontal lines). For BGQ, the percentages indicate the ratios of the numbers of serializations caused by reaching the maximum possible number of rollbacks to the numbers of all the aborts. For Haswell, the percentages are the ratios of the aborts due to HTM buffer overflows to all the aborts. Bolded numbers indicate the points with the minimum runtime per figure. We do not include the numbers for Haswell HLE because it does not enable gathering more detailed statistics [37]. Figures 3d, 3h, and 3l present the total number of HTM events (transactions, aborts, buffer overflows) for every analyzed M.

5.4 Single-vertex Activities

First, we analyze the performance of single-vertex activities. The results are illustrated in Figure 2. Has-C and Has-P follow similar performance trends and we show only the former (denoted as Has); we thus illustrate the results for both a multicore off-the-shelf system and a manycore high performance machine (BGQ).

5.4.1 Activity 1: Marking a Vertex as Visited

Here, each thread uses a CAS or an equivalent HTM code to atomically mark a single vertex; see Fig. 2a-2b, Table 2c. This activity may be used in BFS or any other related algorithm such as Single Source Shortest Path (SSSP). We analyze a negligibly contended scenario that addresses sparse graphs (Fig. 2a; a vertex is marked 10 times to simulate low contention) and a more contended case for dense graphs with high $\bar{d}$ (Fig. 2b; a vertex is marked 100 times). We repeat the benchmark 1000 times and derive the average total time to finish the operations.

Figure 2a shows that Has-CAS finishes fastest and is slightly impacted by the increasing T ($\approx$50% of difference between the results for $T = 4$ and $T = 8$). This is because Has-CAS locks the respective cache line, causing contention in the memory system. Both Has-RTM and Has-HLE have 1.5-3x higher latency than Has-CAS, with Has-RTM being 5-15% faster than Has-HLE. Their performance is not influenced by the increasing T as they rarely abort. Then, BGQ-HTM-S and BGQ-HTM-L are more sensitive to the growing T and their performance drops 11x when switching from $T = 1$ to $T = 64$ due to expensive aborts. As expected, BGQ-HTM-S is faster than BGQ-HTM-L, but as T increases it also aborts more frequently, and becomes $\approx$2x less efficient ($T = 64$) with 37.5% more aborts. BGQ-CAS is least affected by the increasing T.

Figure 2b shows that Has-RTM, BGQ-CAS, BGQ-HTM-S, and BGQ-HTM-L follow similar performance patterns when threads access the vertex 100 times. The performance of Has-HLE drops rapidly as it always performs the costly serialization after the first abort and thus forces all other transactions to abort. The latency of Has-CAS grows proportionally to the contention in the memory system. It stabilizes at $T = 8$ as for $T > 8$ no more operations can be issued in parallel.

5.4.2 Activity 2: Incrementing Vertex Rank

This activity can be used to implement PR. Here, each thread increments the rank of a single vertex 10 times (Figure 2d) and 100 times (Figure 2e) with an ACC or an equivalent HTM code; see Table 2c for details. The most significant difference between the previous and the current benchmark is that the total time and the number of aborts of Has-RTM and Has-HLE grow very rapidly in both scenarios as T scales. This is because in the HTM implementation of ACC, the rank of the vertex is modified by each transaction, generating a considerable number of conflicts and thus aborts. On the contrary, the HTM implementation of CAS generates few memory conflicts: once the vertex id is swapped, other threads only read it and do not modify it. BGQ-HTM-S and BGQ-HTM-L follow a similar trend, with $\approx$3x more aborts than in the previous CAS benchmark.

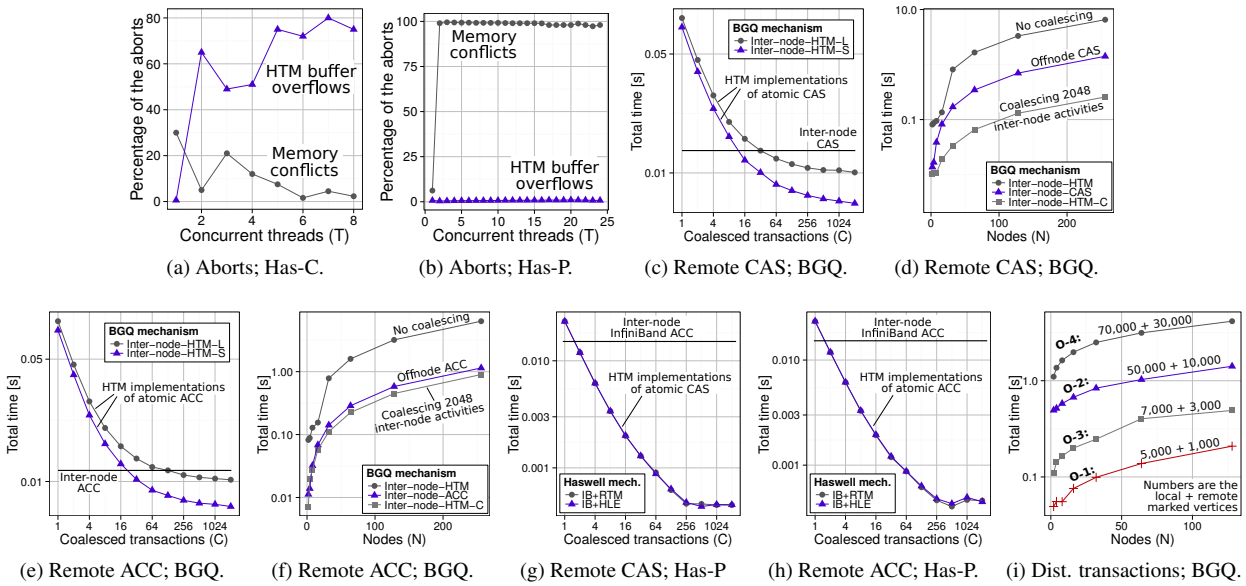
Discussion We present the details of the above analysis in Tables 2c and 2f. We show that the considered single-vertex activities are in most cases best implemented with atomics. HTM is more performant only in processing dense graphs with algorithms that use CAS (e.g., BFS) on Haswell. We also conclude that while atomic CAS is more expensive than ACC, HTM implementation of single ACC is slower ($\approx$100x for RTM and $\approx$10x for BG/Q HTM) than that of CAS as it generates more memory conflicts and thus costly aborts.

5.5 Multi-vertex Activities

The performance analysis of single-vertex intra-node activities illustrates that in most cases a transaction modifying a single vertex is slower than an atomic operation. We now analyze if it is possible to amortize the cost of starting and aborting transactions by enlarging their size, i.e., *coarsening*. This section extends the model analysis (§ 5.3) by introducing effects such as memory conflicts or HTM buffer overflows. We perform the analysis for the highly-optimized OpenMP BFS Graph500 code [27]. We modify the code so that a single transaction atomically visits M vertices and we evaluate the modified code for M between 1 and 320 with the interval of 16. We present the results for three scenarios: a single-threaded execution ($T = 1$ for BGQ, Has-C, and Has-P), a single thread per core ($T = 16$ for BG/Q, $T = 4$ for Has-C, and $T = 12$ for Has-P), and a single thread per SMT hardware resource ($T = 64$ for BG/Q, $T = 8$ for Has-C, and $T = 24$ for Has-P). We use Kronecker graphs [21] with the power-law vertex degree distribution and $|V| = 2^{20}, |E| = 2^{24}$. The results are shown in Figure 3.

5.5.1 BG/Q (Supercomputer)

Figures 3a-3d present the analysis for BG/Q. For $T = 1$ the runtime of both HTM-Long-Mode and HTM-Short-Mode is always higher than that of Atomic-CAS and it decreases initially with the increasing M because higher M reduces the number of transactions required to process the whole graph and thus amortizes the overhead of starting and



(a) Aborts; Has-C. (b) Aborts; Has-P. (c) Remote CAS; BGQ. (d) Remote CAS; BGQ.

(e) Remote ACC; BGQ. (f) Remote ACC; BGQ. (g) Remote CAS; Has-P (h) Remote ACC; Has-P. (i) Dist. transactions; BGQ.

Figure 5: (§ 5.5 & § 5.6) The comparison of the percentage of the reasons of aborts on Has-C and Has-P (Figures 5a and 5b) and the analysis of the performance of inter-node activities on BG/Q and Has-P. The results for BG/Q are following: marking a remote vertex as visited (Figures 5c and 5d), incrementing a vertex' rank (Figures 5e and 5f), and executing distributed transactions (Figure 5i). The results for Haswell are following: marking a remote vertex as visited (Figure 5g) and incrementing a vertex' rank (Figure 5h).

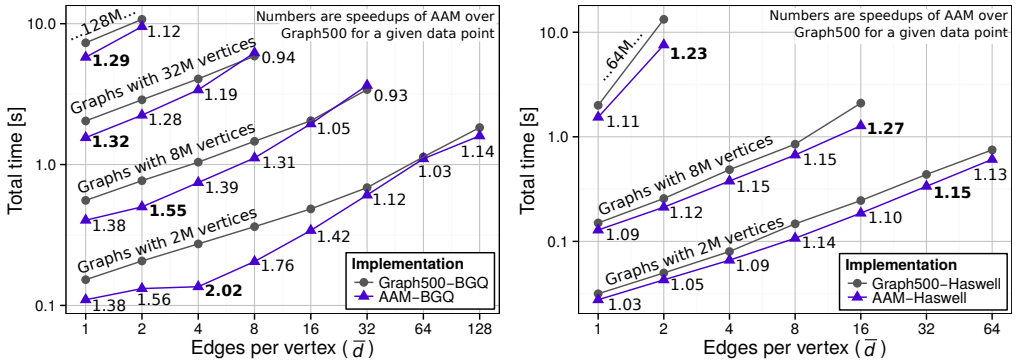
committing transactions. The runtime of `HTM-Short-Mode` becomes higher with the increasing $M > 32$ because this mode is better suited for short transactions. The runtime of `HTM-Long-Mode` decreases as expected and it stabilizes at $M \approx 240$. For $T = 16$, initially the runtime drops rapidly for both HTM modes to reach the minimum (obtained for $M_{min} = 80$ in `HTM-Short-Mode`). Again, this effect is caused by amortizing the overheads of commits/aborts with coarser transactions. Beyond M_{min} the runtime slowly increases with M due to more frequent serializations caused by reaching the maximum number of allowed rollbacks (BGQ does not enable gathering more detailed statistics but we predict that these serializations are due to the higher number of HTM buffer overflows and memory conflicts). `HTM-Long-Mode` is never more performant than `Atomic-CAS`. `HTM-Short-Mode` becomes more efficient than `Atomic-CAS` for $M = 32$ and achieves the speedup of 1.11 at $M_{min} = 80$. A similar performance pattern can be observed for $T = 64$; this time $M_{min} = 144$ in `HTM-Short-Mode` with the speedup of 1.49 over `Atomic-CAS`. The runtime becomes dominated by aborts for $M > 144$; cf. Figure 3d with more detailed numbers of aborts.

5.5.2 Has-C (Commodity Machine)

The results of the analysis for Has-C are presented in Figures 3e-3h. In each scenario ($T = 1, 4, 8$) the performance of both `HTM-RTM` and `HTM-HLE` decreases with increasing M. Several outliers are caused by disadvantageous graph data layouts that entail more aborts due to the limited associativity of L1 cache (8-way associative cache) that stores speculative states in Haswell [37]. We perform a more detailed analysis for $M \in \{1, ..., 16\}$ to find out that $M_{min} = 2$. `HTM-RTM` becomes less performant than `HTM-HLE` at M≈200 because the cost of serializations due to the HTM buffer overflows dominates the runtime of `HTM-RTM` beyond this point (serializations in `HTM-HLE` are implemented in hardware [37], while in `HTM-RTM` they have to be implemented in software).

5.5.3 Has-P (High-Performance Server)

The analysis of Has-P is presented in Figures 3i-3l. The performance trends are partially similar to the observations for Has-C; especially for lower thread counts ($T \leq 4$). A distinctive feature is a significantly lower number of HTM buffer overflows than in Has-C. To gain more insight we performed an additional analysis to compare the number of memory conflicts and HTM buffer overflows with varying T for fixed $M = 2$. We present the results in Figures 5a-5b. Surprisingly, we observe Has-C has significantly more buffer overflows than memory conflicts for the increasing T; a reverse trend is observed on Has-P. This interesting insight may help improve the design of future HTM architectures.



(a) The performance of BFS on BG/Q for Kronecker graphs.(b) The performance of BFS on Haswell for Kronecker graphs.

Figure 6: (§ 6.1) The overview of the performance of intra-node Graph500 BFS implemented with atomics (`Graph500-BGQ`, `Graph500-Haswell`), AAM RTM (`AAM-Haswell`), and the short mode BG/Q HTM (`AAM-BGQ`). We vary $|V|$ and $\bar{d}$; $T = 64$ (on BG/Q) and $T = 8$ (on Has-C).

Discussion Our analysis shows that RTM is more vulnerable to aborts than BG/Q HTM. The difference between the number of transactions and aborts never drops below 25% for HTM in BG/Q for any analyzed M (cf. Figure 3d), while for RTM this threshold is achieved for $M = 144$ (Has-C). Another discovery is that Has-P is only marginally impacted by buffer overflows (¡1% of all the aborts for $T = 24$ and $M = 320$). On the contrary, aborts in Has-C are dominated by HTM buffer overflows that constitute more than 90% of all the aborts for $M > 64$. The only exception are the data points where the number of overflows drops rapidly as aborts become dominated by the limited L1 cache associativity (a similar effect is visible for Has-P). This effect is not visible in BG/Q because it stores its speculative states in its L2 16-way associative cache [33], while both Has-P and Has-C have 8-way associative L1s.

We conclude that the coarsening of transactions provides significant speedups (up to 1.51) over the `Atomic-CAS` baseline on BGQ and Has-C; Has-P does not offer any speedups due to the overheads generated by memory conflicts. We find the following optimum transaction sizes for PowerPC in BG/Q: $M_{min} = 80$ ($T = 16$), $M_{min} = 144$ ($T = 64$). For `x86` (Has-C) $M_{min} = 2$ for $T \in \{4, 8\}$. We will use these values in Section 6 to accelerate Graph500 [27] for different types of graphs.

5.6 Activities Spawned on a Remote Node

We now analyze the performance of activities spawned on a remote node. We implement such activities as hardware transactions triggered upon receiving an atomic active message. We again test both the long and the short running mode (on BG/Q) and RTM/HLE (on Haswell). To reduce the overhead of sending and receiving an atomic active message and save bandwidth, we use *activity coalescing*: activities flowing to the same target are sent in a single message.

We run the benchmarks on BG/Q and Greina (Has-P); we skip Has-C because the Trivium server is not a distributed memory machine. On BG/Q, we compare inter-node activities to optimized remote one-sided CAS and ACC atomics provided by the generic function `PAMI_Rmw` in the IBM PAMI communication library [17]. On Has-P we compare activities to remote atomic operations provided by MPI-3 RMA [26] implemented over the InfiniBand fabric. We evaluate the performance of marking a remote vertex as visited (addressing distributed BFS computations) and incrementing the rank of a remote vertex (addressing distributed PageRank).

5.6.1 BG/Q (Supercomputer)

We first measure the time it takes a process p_i to mark 2^{13} vertices stored on a node n_j as visited (targeting distributed BFS). The results are presented in Figure 5c. Without coalescing, HTM activities (`Inter-node-HTM-L` for the long and `Inter-node-HTM-S` for the short mode) are ≈5x slower than PAMI atomics (`Inter-node-CAS`). Still, for $C_{cross} = 16$ `Inter-node-HTM-S` becomes more performant. Second, we scale the number of nodes N. Figure 5d shows the time to mark a vertex stored in process p_N's memory by $N-1$ other processes. We use the short HTM mode. Coalesced AAMs (`Inter-node-HTM-C`) outperform `Inter-node-CAS` ≈5-7 times.

We also evaluate an inter-node activity that increments the rank of a vertex (targeting distributed PR). We perform analogous benchmarks as for the remote CAS; we present the results in Figures 5e-5f. Implementing ACC

using HTM again generates costly aborts that dominate the runtime; however coalescing enables a speedup of ≈20% (for the short HTM running mode) over highly optimized PAMI atomics.

5.6.2 Has-P (High-Performance Server)

Here, we test the performance of inter-node activities implemented on Has-P. Our testbed has two nodes, thus we only vary C. We present the results in Figures 5g (CAS) and 5h (ACC). Here, setting $C = 2$ enables AAM to outperform remote atomics provided by MPI-3 RMA.

5.7 Distributed Activities

Finally, we test the ownership protocol for executing activities that span multiple nodes (see Figure 5i for BGQ results). Here, each process issues x transactions; each transaction marks a local and b remote randomly selected vertices. We compare four scenarios: `O-1` ($x = 10^3, a = 5, b = 1$), `O-2` ($x = 10^4, a = 5, b = 1$), `O-3` ($x = 10^3, a = 7, b = 3$), and `O-4` ($x = 10^4, a = 7, b = 3$). We measure the total time to execute transactions. `O-1` finishes fastest, `O-3` is slower as more remote vertices have to be acquired. `O-2` and `O-4` follow the same performance patterns; additional overheads are due to the backoff scheme. If no time is spent on backoff, then the protocol may livelock and may make no progress.

We conclude that AAM can be used in various environments (e.g., IBM networks or InfiniBand) to enable remote transactions and to accelerate distributed processing.

6 Evaluation

We now use AAM to accelerate the processing of large Kronecker [21] and Erdős-Renyi [9] (ER) graphs with different vertex distributions (power-law, binomial, Poisson). We also evaluate real-world SNAP graphs[2]. We evaluate BFS and PR because they are the basis of various data analytics benchmarks such as Graph500 and because they are proxies of many algorithms such as Ford-Fulkerson.

6.1 BFS: Massively-Parallel Manycores

We first evaluate the speedup that AAM delivers in highly-parallel multi- and manycore environments.

Comparison Baseline: Here, we use the OpenMP Graph500 highly optimized reference code [27] (`Graph500-BGQ`, `Graph500-Haswell`) based on atomics as the comparison baseline. The baseline applies several optimizations; among others it reduces the amount of fine-grained synchronization by checking if the vertex was visited before executing an atomic.

We compare the Graph500 baseline with the coarsened variants that use the short mode HTM in BG/Q (`AAM-BGQ`) and RTM in Haswell (`AAM-Haswell`). Here, we only use Has-C (denoted as Haswell) because it provides higher speedups over atomics than Has-P as we show in Figure 3. The long mode and HLE are omitted as they follow similar performance patterns and vary by up to 10%. We set $T = 64$ (for BG/Q) and $T = 8$ (for Haswell) for full parallelism.

6.1.1 Processing Kronecker Power-Law Graphs

Here, we use the results of the analysis in Section 5 and set $M_{min} \in \{2, 80, 144\}$ for the most advantageous size of transactions on BG/Q and Haswell. We present the results in Figure 6. We scale $|V|$ from 2^{20} to 2^{28}, and we use $\bar{d} \in \{1, 2, ..., 256\}$; highest values generate graphs that fill the whole available memory. For BG/Q, `AAM-BGQ` outperforms `Graph500-BGQ` by up to 102% for a graph with ≈2 millions vertices and $\bar{d} = 4$. For higher $\bar{d}$ `AAM-BGQ` becomes comparable to `Graph500-BGQ`. This is because adding more edges for fixed $|V|$ generates more transactions that conflict and abort more often. For Haswell, AAM consistently outperforms Graph500 by up to 27%. The speedup does not change significantly when increasing $\bar{d}$. This is because we use smaller transactions in `AAM-Haswell` ($M = 2$) than in `AAM-BGQ` ($M = 144$) and thus they do not incur considerably more memory conflicts when $\bar{d}$ is increased.

6.1.2 Processing Real-World Graphs

Next, we evaluate AAM for real-world graphs (see Table 1). For this, we extend Graph500 so that it can read graphs from a file. We selected directed/undirected graphs with $|V| > 250k$ that could fit in memory and we excluded graphs that could not easily be loaded into Graph500 framework (e.g., amazon0505).

BlueGene/Q: The tested graphs are generally sparser than the analyzed Kronecker graphs. We discovered that the optimum M is smaller than 144 (we set it to 24). This is because in dense graphs more data is contiguous in memory and thus can be processed more efficiently by larger transactions. The results show that graphs with similar structure entail similar performance gains. The highest S (speedup) is achieved for `CN`s (up to 3.67) and `WG`s (up to 1.91). `SN`s, `PN`s, and `CG`s offer moderate S (1.14-1.67). `RN`s entail no significant change in performance. We also searched for optimum values of M for specific graphs; this improves S across all the groups. The results indicate that respective groups have

[2] Available at `https://snap.stanford.edu/data/index.html`.

Input graph properties					BG/Q analysis			Haswell analysis						
Type	ID	Name	$\|V\|$	$\|E\|$	S over g500 ($M=24$)	M	S over g500	S over g500 ($M=2$)	S over Galois ($M=2$)	M	S over g500	S over Galois	S over HAMA	S over SNAP
Comm. networks (CNs)	cWT	wiki-Talk	2.4M	5M	2.82	48	3.35	0.91	1.22	6	0.96	1.28	344	$>10^4$
	cEU	email-EuAll	265k	420k	3.67	32	4.36	0.76	0.88	4	0.97	1.12	1448	273
Social networks (SNs)	sLV	soc-LiveJ.	4.8M	69M	1.44	12	1.56	1.05	1.1	3	1.07	1.12	$>10^4$	$>10^4$
	sOR	com-orkut	3M	117M	1.22	20	1.27	1.06	0.69	4	1.13	0.74	$>10^4$	$>10^4$
	sLJ	com-lj	4M	34M	1.44	12	1.54	1.03	1.03	4	1.04	1.04	603	$>10^4$
	sYT	com-youtube	1.1M	2.9M	1.67	8	1.84	0.96	1.1	5	0.98	1.11	670	6164
	sDB	com-dblp	317k	1M	1.33	8	1.80	$\approx$1	2.5	2	$\approx$1	2.53	2160	2172
	sAM	com-amazon	334k	925k	1.14	8	1.62	1.04	1.64	2	1.04	1.64	1426	1356
Purchase network (PNs)	pAM	amazon0601	403k	3.3M	1.45	8	1.91	$\approx$1	1.25	3	1.03	1.30	618	2878
Road networks (RNs)	rCA	roadNet-CA	1.9M	5.5M	$\approx$1	2	1.59	1.33	1.74	8	1.38	1.80	$>10^4$	$>10^4$
	rTX	roadNet-TX	1.3M	3.8M	$\approx$1	2	1.53	1.29	1.89	6	1.42	2.08	$>10^4$	$>10^4$
	rPA	roadNet-PA	1M	3M	$\approx$1	2	1.52	$\approx$1	2.00	9	1.07	2.16	$>10^4$	$>10^4$
Citation graphs (CGs)	ciP	cit-Patents	3.7M	16.5M	1.16	8	1.57	1.01	1.26	2	1.01	1.26	1875	$>10^4$
Web graphs (WGs)	wGL	web-Google	875k	5.1M	1.78	12	2.08	0.98	1.26	6	1.06	1.35	365	3950
	wBS	web-BerkStan	685k	7.6M	1.91	24	1.91	0.93	1.31	5	1.07	1.40	755	7125
	wSF	web-Stanford	281k	2.3M	1.89	24	1.89	0.98	1.54	5	1.07	1.58	1077	1570

Table 1: (§ 6.1.2) The performance of AAM for real-world graphs. S and g500 denote speedup and Graph500. $\approx$1 indicates that the given $S \in (0.99; 1.01)$.

similar optimum values of M. The differences are due to the structures of the graphs that may either facilitate coarsening and reduce the number of costly aborts (CNs and WGs) or entail more significant overheads (RNs).

Haswell: Here, we compare AAM to several state-of-the-art graph processing engines: SNAP [20] (represents network analysis and data mining libraries), Galois [16] (represents runtime systems that exploit amorphous data-parallelism), and HAMA [30] (an engine similar to Pregel [24] that represents Hadoop-based BSP processing engines). We do not evaluate these engines on BG/Q due to various compatibility problems (e.g., BG/Q does not support Java required by HAMA). BFS in Galois only returns the diameter. We modified it (with fine locks) so that it constructs a full BFS tree, analogously to AAM and Graph500.

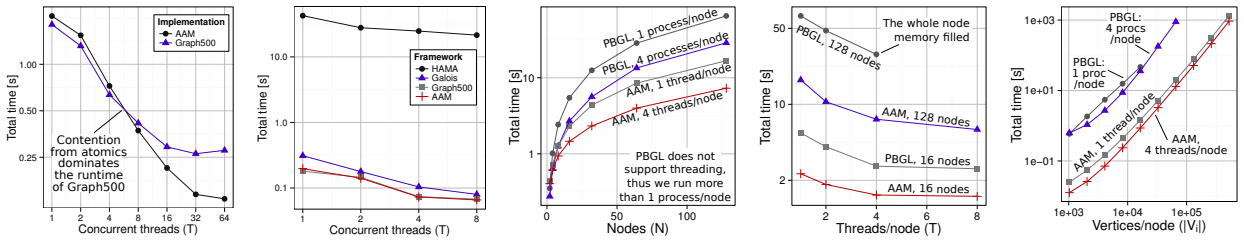
First, we set $M = 2$. While AAM is in general faster than Graph500 (up to 33% for rCA), several inputs entail longer AAM BFS traversals. AAM is up to a factor of two faster than Galois but is slower for two inputs (cEU and sOR). There is some diversity in the results because AAM on Haswell is significantly more sensitive to small changes of M than on BG/Q. Thus, we again searched for the optimum M for each input separately which resulted in higher AAM's speedups. The performance of HAMA and SNAP is generally much lower than AAM. HAMA suffers from overheads caused by the underlying MapReduce architecture and expensive synchronization. The analyzed real-world graphs have usually high diameters (e.g., 33 for sAM) and thus require many BSP steps that are expensive in HAMA. This is especially visible for RNs that have particularly big diameters (554 for rCA) and accordingly long runtimes. As we will show in the next section, processing Kronecker graphs with lower diameters reduces these overheads. We also investigated SNAP and we found out that it is particularly inefficient for undirected graphs and it does not efficiently use threading. Our final discovery is that, similarly to BG/Q, respective groups of graphs have similar optimum values of M.

6.1.3 Evaluating the Scalability of AAM

Finally, we evaluate the scalability of AAM by varying T. The results are presented in Figure 7a (BG/Q) and 7b (Haswell). We use a Kronecker graph with 2^{21} vertices and 2^{24} edges. We vary T between 1 and the number of available hardware threads. The BG/Q results indicate that AAM utilizes onnode parallelism more efficiently than Graph500. For Haswell, the performance patterns for AAM and Graph500 are similar; both frameworks deliver positive speedups for any T and outperform other schemes by $\approx$20-50% (Galois) and $\approx$2 orders of magnitude (HAMA). We skip SNAP for clarity; it is consistently 2-3x slower than HAMA.

6.2 PR: Distributed Memory Machines

As the last step, we provide an initial large-scale evaluation of AAM in a distributed environment. We select PR to illustrate that expensive and numerous aborts generated by the HTM implementation of ACC (cf. § 5.4.2) can be amortized with the coalescing of activities. We compare AAM to a version of Parallel Boost Graph



(a) Scalability of AAM; BG/Q. (b) Scalability of AAM; Haswell. (c) PageRank, $ER = 0.0005$. (d) PageRank, $ER = 0.0005$. (e) PageRank, $ER = 0.005$.

Figure 7: The analysis of the performance of BFS when varying T (§ 6.1.3, Figures 7a-7b), and the overview of the performance of distributed PR (§ 6.2, Figures 7c-7e).

Library (PBGL) [10] based on active messages. The utilized variant of PBGL applies various optimizations; for example it processes incoming edges to reduce the amount of synchronization and to limit the performance overheads caused by atomics. We run the benchmarks on BG/Q to enable large-scale evaluation. We use ER graphs with the probability parameter $ER \in \{0.005, 0.0005\}$ and the number of vertices up to 2^{23}. PBGL does not support threading, we thus spawn multiple processes per node and an equal number of threads in AAM; we scale T until PBGL fills in the whole memory.

The results of the analysis are presented in Figure 7. We scale N (Figure 7c), T (Figure 7d), and $|V_i|$ (Figure 7e). In each scenario AAM outperforms PBGL ≈3-10 times thanks to the coalescing of activities and more efficient utilization of intra-node parallelism.

7 Related Work and Discussion

The challenges connected with the processing of graphs are presented by Lumsdaine et al. [23]. Example frameworks for parallel graph computations are Pregel [24], PBGL [10], HAMA [30], GraphLab [22], and Spark [38]. A recent comparison of various engines was done by Satish et al. [29]. AAM differs from these designs as it is a mechanism that can be used to implement abstractions and to accelerate processing engines. It uses HTM to reduce the amount of synchronization and thus to accelerate graph analytics.

GraphBLAS [25] is an emerging standard for expressing graph computations in terms of linear algebra operations. AAM can be used to implement the GraphBLAS abstraction and to accelerate the performance of graph analytics based on sparse linear algebra computations.

The Galois runtime [15] optimizes graph processing by coarsening fine graph updates. AAM can be integrated with Galois. In AAM, we focus on scalable techniques for implementing coarsening with HTM. First, we provide a detailed performance analysis of HTM for graph computations, a core paper contribution. Instead, Galois mostly addresses locking [16]. Second, contrary to Galois, AAM targets both shared- and distributed-memory systems. Third, our work performs a holistic extensive performance analysis of coarsening. Instead, coarsening in Galois is not evaluated on its own. We conclude that AAM's techniques and analysis can be used to accelerate the Galois runtime.

Active Messages (AM) were introduced by Eicken et al. [32]. Various AM implementations were proposed [8, 35, 36, 17, 2]. Our work enhances these designs by combining AM with HTM. We illustrate how to program AAM and we conduct an extensive analysis to show how to tune AAM's performance on state-of-the-art manycore architectures.

Transactional memory was introduced by Herlihy et al. [12]. Several implementations of HTM were introduced, but their performance was not extensively analyzed [37, 33, 7, 6]. Yoo et al. [37] present performance gains from using Haswell HTM in scientific workloads such as simulated annealing. Our analysis generalizes these findings, proposes a simple performance model, and provides a deep insight into the performance of both BG/Q and Haswell HTM for a broad range of transaction sizes and other parameters in the context of data analytics.

Finally, we envision that the potential of AAM could be further expanded by combining it with some ideas related to code analysis. For example, one could envision a simple compiler pass that pattern-matches each single-vertex transaction against the set of atomic operations to transform it if possible to accelerate graph processing. However, such an analysis is outside the scope of this paper.

8 Conclusion

Designing efficient algorithms for massively parallel and distributed graph computations is becoming one of the

key challenges for the parallel programming community [16]. Graph processing is fine-grained by nature and its traditional implementations based on atomics or fine locks are error-prone and may entail significant overheads [16].

We propose Atomic Active Messages (AAM), a mechanism that reduces that amount of fine-grained synchronization in graph analytics. AAM is motivated by recent advances towards implementing transactional memory in hardware. AAM provides several high performance techniques for executing fine-grained graph modifications as coarse transactions; it facilitates the utilization of state-of-the-art hardware mechanisms and resources and can be used to accelerate highly optimized codes such as Graph500 by more than 100%.

AAM targets highly-parallel multi- and manycore architectures and distributed-memory machines. It provides a novel classification of atomic active messages that can be used to design and program both shared- and distributed-memory graph computations. AAM enables different optimizations from both of these worlds such as coarsening intra-node transactions and coalescing inter-node activities. We illustrate how to implement AAM with HTM; however other mechanisms such as distributed STM [19], flat-combining [11], or optimistic locking [18] could also be used.

Finally, to the best of our knowledge, our work is the first detailed performance analysis of hardware transactional memory in the context of graph computations and the first to compare HTMs implemented in Intel Haswell and IBM Blue Gene/Q. Among others, we conjecture that implementing HTM in the bigger L2 cache (BG/Q) enables higher performance than in the smaller L1 cache (Haswell). We believe our analysis and data can be used by architects and engineers to develop a more performant HTM that would offer even higher speedups for irregular data analytics.

We thank Hussein Harake and the whole CSCS team for the access to the Greina and Monte Rosa machines and for their excellent technical support.

References

[1] E. G. Boman, U. Catalyurek, A. Gebremedhin, and F. Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *in Proc. of Euro-Par 2005 Par. Proc.*, pages 241–251, 2005.

[2] D. Bonachea. GASNet Specification, v1. *Univ. California, Berkeley, Tech. Rep. CSD-02-1207*, 2002.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of Intl. Conf. on World Wide Web*, WWW7, pages 107–117, 1998.

[4] A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. *Cont. Math.*, 588, 2013.

[5] C. Click. Azul's experiences with hardware transactional memory. In HP Labs' Bay Area Workshop on Transactional Memory.

[6] S. Chaudhry et al. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, Mar. 2009.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, ASPLOS XIV, pages 157–168, 2009.

[8] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proc. of the ACM Intl. Conf. on Supercomp.*, ICS '13, pages 283–292, 2013.

[9] P. Erdos and A. Rényi. On the evolution of random graphs. *Pub. Math. Inst. Hun. A. Sci.*, 5:17–61, 1960.

[10] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[11] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of the ACM Symp. on Par. in Alg. and Arch.*, SPAA '10, pages 355–364, 2010.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the Intl. Symp. on Comp. Arch.*, ISCA '93, pages 289–300, 1993.

[13] Intel. Intel Xeon Processor E3-1200 v3 Product Family, December 2014. Revision 009.

[14] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *Proc. of the Symp. on Prin. Prac. of Par. Prog.*, PPoPP '09, pages 15–24, 2009.

[15] M. Kulkarni et al. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, PLDI '07, pages 211–222, 2007.

[16] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic Parallelism Benefits from Data Partitioning. In *Proc. of the 13th Intl. Conf. on Arch. Sup. for Prog. Lang. and Op. Sys.*, ASPLOS XIII, pages 233–243, 2008.

[17] S. Kumar et al. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *IEEE Par. Dist. Proc. Symp. (IPDPS)*, pages 763–773, 2012.

[18] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[19] M. Lesani and J. Palsberg. Communicating memory transactions. In *Proc. of the Symp. on Prin. and Prac. of Par. Prog.*, PPoPP '11, pages 157–168, 2011.

[20] J. Leskovec. *Dynamics of large networks*. PhD thesis, Carnegie Mellon University, 2008.

[21] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[22] Y. Low et al. Graphlab: A new framework for parallel machine learning. *preprint arXiv:1006.4990*, 2010.

[23] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Par. Proc. Let.*, 17(1):5–20, 2007.

[24] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Manag. of Data*, SIGMOD '10, pages 135–146, 2010.

[25] T. Mattson et al. Standards for Graph Algorithm Primitives. *CoRR*, abs/1408.0393, 2014.

[26] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3, September 2012.

[27] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[28] O. Reingold. Undirected ST-connectivity in Log-space. In *Proc. of the ACM Symp. on Theory of Comp.*, STOC '05, pages 376–385, 2005.

[29] N. Satish et al. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 979–990, 2014.

[30] S. Seo et al. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Proc. of the IEEE Intl. Conf. on Cloud Comp. Tech. and Science*, CLOUDCOM'10, pages 721–726, 2010.

[31] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of ACM Symp. on Pr. of Dist. Comp.*, PODC '95, pages 204–213, 1995.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Intl. Symp. on Comp. Arch.*, ISCA '92, pages 256–266, 1992.

[33] A. Wang et al. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proc. of the Intl. Conf. on Par. Arch. and Comp. Tech.*, PACT '12, pages 127–136, 2012.

[34] J.-T. W. S. Weigert. Dream: Dresden streaming transactional memory benchmark. 2013.

[35] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. AM++: A Generalized Active Message Framework. In *Proc. of the Intl. Conf. on Par. Arch. and Comp. Tech.*, pages 401–410, Sep. 2010.

[36] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active Pebbles: Parallel Programming for Data-Driven Applications. In *Proc. of the ACM Intl. Conf. on Supercomp. (ICS'11)*, pages 235–245, 2011.

[37] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proc. of the ACM/IEEE Supercomputing*, SC'13, 2013.

[38] M. Zaharia et al. Spark: Cluster Computing with Working Sets. In *Proc. of the USENIX Conf. on Hot Top. in Cl. Comp.*, HotCloud'10, pages 10–10, 2010.